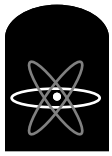
UCRL-ID-112616
NRC FIN L-1867

Real-Time Performance

G.G. Preckshot

May 28, 1993



FESSP
Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

Disclaimer

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy.

Real-Time Performance

G.G. Preckshot

Manuscript Date: May 28, 1993

Contents

Real-Time Performance

Executive Summary

This paper has three specific goals for regulatory guidance. The first is to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second goal is to present and recommend timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third is to suggest means for assessing expected or actual real-time performance before, during, and after development is completed.

The thesis of the paper is that safety-critical real-time systems must be predictable in order to satisfy the three goals. In this paper, technical background is provided on methods of timing analysis, scheduling real-time computations, prototyping, real-time software development approaches, modeling and measurement, and real-time operating systems. The background material supports recommendations that guide reviewers and developers in prototype testing, timing analyses, and ensuring that real-time performance requirements are met.

Research results developed for prototype testing suggest that three kinds of prototypes will be useful, either for developing or reviewing nuclear reactor safety software. These are human factors prototypes, early prototypes to resolve technical questions, and validation prototypes for performance estimation of the finished software product. Performance models are useful as a guide for determining the scale of the final prototype.

Work in the literature on timing analyses suggests that, with certain restrictions, it is possible to predict lower and upper bounds for code execution time. This ability is a prerequisite for several known methods of producing predictable real-time systems. If the restrictions are not met, then any demonstration of predictability should be at least as convincing as the proof for an equivalent restricted system.

To ensure that the delivered real-time software product meets performance goals, this paper covers certain types of code-execution and communications scheduling that can be demonstrated to be correct and to meet deadlines. If these types of scheduling are not used, then alternative proofs should be at least as rigorous as the described methods, or doubts will exist that the delivered system will perform under all circumstances.

Finally, current practice in the use of performance models is discussed. Performance models and measured performance should converge, and prototype testing should demonstrate that system performance scales as indicated by the models.

1. Introduction

Real-time computer systems are now used in some nuclear reactor control systems and protection systems. Recent failures (the French Controbloc P20) and validation difficulties (Ontario Hydro's Darlington Plant) have focused attention on system performance and software reliability determination. This paper reviews methods for building real-time computer systems for reactor protection systems to meet performance goals, and offers recommendations for regulators who must determine if appropriate methods were followed during development, whether the system design is likely to meet performance requirements, and what testing or prototype testing should be done to verify this.

1.1 Goals

This paper has three specific goals for regulatory guidance. The first is to determine if, when, and what prototypes should be required of developers to make a convincing demonstration that specific problems have been solved or that performance goals have been met. The second goal is to present and recommend timing analyses that will prove that the real-time system will meet its safety-imposed deadlines. The third is to suggest means for assessing expected or actual real-time performance before, during, and after development is completed.

1.1.1 Need for Prototype Testing

The computer science meaning of the word "prototype" is not only double-faceted but differs from the traditional engineering use of the term. Engineers mean "first of type" or "first off the line," and an engineering prototype is normally used to perform system tests to verify either design correctness or manufacturability. In contrast, software engineers use prototypes to demonstrate system–user interactions to prospective users, or to determine if specific design approaches will work. The former software engineering use is to refine user requirements, while the latter is to demonstrate that there are feasible ways to meet the requirements.

Of the three kinds of prototypes, software engineering feasibility prototypes and traditional engineering prototypes are most applicable to real-time systems for nuclear reactor protection systems. This will be discussed at more length in Section 3.4.

1.1.2 Timing Analyses

Timing analyses are performed to reduce the probability that expected or unexpected combinations of data values or event sequences will cause real-time systems to miss deadlines. A deadline, in this context, is a time limit on a real-time system's response to specific stimuli. If timing analyses are not performed, then real-time software remains a black box whose deadline performance in all situations is uncertain.

There are several kinds of timing analysis, of which three types will be discussed in this report. The simplest, and most obvious, is computing expected code run time by adding up individual instruction execution times. This method works well for code modules, but does not address the timing of module interactions, which is the second analysis method addressed. The third kind of analysis, intermodule sequencing, is not directed at absolute timing at all, but at whether code modules execute in correct order of precedence. The three methods can be considered, respectively, as module timing analysis, system timing analysis, and schedulability and priority analysis.

1.1.3 Assessing Real-Time Performance

A naíve approach to assessing real-time system performance suggests that the completed system should be tested under a selection of work loads, and performance should be deduced in the manner of final acceptance testing of electrical machinery. Unfortunately, the design of distributed, interconnected computer systems admits far more complexity and variability than traditional electric machine design, so that performance assessment is a process that begins in the initial system design stages. The difficulty is that there is no commonly accepted system model as there is with electrical machinery. Consequently, unless a specifically tailored system model is developed while the system is being designed, it is very difficult to devise performance tests and work loads that adequately measure realistic system performance.

Two general kinds of system model are popular—queuing models and Petri net models. Each has been enhanced by various investigators in attempts to compensate for real deficiencies in the models, with the result that a possibly confusing number of subtypes now exist. Each model benefits from data produced by timing analyses, and is in turn used in later stages of system timing analysis for simulation and analytic modeling. The usual progression, which will be elaborated later, is initial modeling to project expected system performance, model refinement and use for development guidance during intermediate development stages, and final resolution of conflicts and misunderstandings during performance testing and system validation. Ideally, at the end of the process, the reviewer, the developer, the model, and the system produced will all agree.

1.2 Structure of This Paper

The balance of this paper addresses the above goals. However, before discussing the scope of the paper's coverage (Section 2.4), a number of general issues about real-time systems, including definition, are introduced to make succeeding comments about scope intelligible. After delineating scope, a technical background section introduces predictability, timing, and scheduling, and then considers prototyping, development approaches, and modeling and measurement. A short subsection on a limited set of real-time operating systems is provided to introduce pertinent characteristics of such products. Finally, recommendations and guidelines are provided for prototyping, timing analyses, and assessing real-time performance of candidate systems.

2. Real-Time Systems Defined

Real-time systems are computer systems which must produce correct outputs in response to real-world inputs within well-defined deadlines (Burns and Wellings 1990). This definition places an additional correctness constraint on software for real-time systems: even if the values produced by software are correct, the software is considered to fail if it does not meet its deadline. Current-day practice further divides real-time systems into hard real-time systems and soft real-time systems. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and will still function correctly. An example of the former is any real-time aircraft or missile flight control system. A deadline failure in either of these systems could cause loss of aircraft or missile. Airline flight reservation systems are examples of soft real-time systems. An occasional missed deadline simply delays a reservation clerk and irritates customers. Current research in real-time systems has identified a number of important issues and misconceptions. It is popularly assumed that fast computing is real-time

computing. This is untrue. While speedy computers undoubtedly help by reducing computation time, real-time systems are concerned with meeting deadlines rather than going as fast as possible (Stankovic 1988). This also accounts for the difference between "performance engineering" and real-time systems, with the former using statistical performance indices that permit variability that the latter cannot tolerate.

A second important theme illustrated by the difference between the conventional performance engineering design approach and real-time system design is predictability. Real-time system developers seek to guarantee timing and schedulability of tasks to ensure that the real-time product meets deadlines with absolute predictability (Natarajan and Zhao 1992). There are two approaches to this problem, and they can be characterized as static versus dynamic scheduling. A statically scheduled real-time system has a schedule or schedules that have been previously calculated, while in a dynamically scheduled system, some or all of the tasks are scheduled at unpredictable times by a real-time operating system or dedicated scheduler. Even though tasks are scheduled at unpredictable times in dynamically scheduled systems, it still may be possible to predict that deadlines will be met for a subset of tasks. This is usually more difficult to do than it would be for a statically scheduled system.

Statically scheduled systems are typically designed for worst-case load, and therefore may use more resources than dynamically scheduled systems. Resources include CPU time, memory, communication link bandwidth, and the use of various devices such as printers, keyboards, displays (CRTs), and the like. The advantage of statically scheduled systems is that, for real-time applications that have a well-known rota of functions to accomplish, it is possible to produce software systems with very convincing guarantees of deadline performance. The relative simplicity of the approach also has merit for safety- or life-critical systems.

Static systems do not do well in applications where tasks are not fixed—for example, systems with unpredictable function demands or unpredictable computations (recursions, convergence, data-dependent algorithms) which may extend execution times beyond planned limits. In these cases, dynamically scheduled systems can provide more functional flexibility and better resource usage, at the expense of more difficult solution of resource requirements, availability, and timing guarantee problems. The hardest problem is demonstrating that dynamically scheduled systems will handle overloads and faults acceptably, since these systems are not designed or sized for worst-case operation.

Both statically and dynamically scheduled systems will be considered in following sections. Scheduling is central to current real-time systems theory, and the subject is interwoven through many of the subjects to be treated.

2.1 Complexity

A significant issue for the reviewer is the complexity of a candidate real-time system and how it will affect the difficulty of ensuring adequate performance. In a previous paper (reprinted in this report as Appendix B) an approximate complexity scale was introduced, and it is instructive to consider that scale in light of what has just been written about predictability and scheduling. For convenience, the complexity attributes are repeated below:

- Timing complexity.
- Priority complexity.
- Resource complexity.
- Connection complexity.
- Concurrency.
- Communication delay.

In general, statically scheduled systems are progressively more difficult to schedule as progressively more complexity attributes are present in the design. However, a static schedule, once made, can be analyzed for correctness and deadline conformance. Since the schedule calculations are made off-line, the computational expense and delay incurred by lengthy scheduling algorithms are not factors in system performance. Developers therefore have considerable latitude in tools used to produce an acceptable task and communication schedule, and more complex system interactions may be provably safe and timely.

In contrast, dynamically scheduled systems exhibit at least timing complexity and, if resources are shared between tasks and preemption is allowed, there is priority and resource complexity as well. Unlike statically scheduled systems, where it is only necessary to prove *a posteriori* that the schedule is correct, here it must be proven *a priori* that an unknown schedule will be correct. Priority and resource interactions considerably complicate this problem. The addition of connection complexity (intertask communication), concurrency, and communication delay may make definitive proofs of correctness and deadline performance impossible without significant restrictions on the software design.

2.2 State-Based Systems

Another important distinction that applies to real-time systems is whether they are state-based or event-driven systems. These are idealizations of two approaches to overall design of real-time systems. Differences between the two have been elaborated by Kopetz (circa 1990) and discussed with regard to communication systems by Preckshot (1993). State-based systems transmit "state" (data that describe external and internal conditions) at regular intervals and recalculate control outputs after each state exchange. Because most state information is unchanging or slowly changing, this method results in inefficient use of communication resources and CPU bandwidth. However, the system is not subject to overload because the same amount of data is transmitted no matter what the circumstances are. The elimination of the overload performance problem is often regarded as ample recompense for less efficient use of resources, and state-based systems are preferred in ultra-reliable applications.

2.3 Event-Driven Systems

Event-driven systems transmit information only when "significant" events occur and recalculate control outputs at event times. Leaving aside how "significant" is defined for the moment, this method results in less resource usage during quiet periods, but may cause system overload and failure to respond when control or protective action is most needed. Event-driven systems are inappropriate for applications where regular, precisely timed control outputs are required (for example, closed-loop control), but are often used where intermittent, non-linear actions are required (such as contact closures to start or stop motors). Hybrid event-driven and state-based systems are possible.

The question of what is "significant" is sometimes crucial to system operation. Too strict a bound on significance (e.g., change in value required is too large) may result in the system being insensitive to

something it should see. Too loose a bound (e.g., small changes in value are "significant") may result in frequent overloads and degraded performance. In general, the performance of event-driven systems is more difficult to predict than state-based systems (Kopetz circa 1990). Therefore, some event-driven or hybrid systems may be unacceptable for safety applications because a final safety determination is too difficult or is impossible to make.

2.4 Scope

Reviewers of safety-critical software are unlikely to be satisfied with unpredictable real-time systems used in safety-critical systems, and are not pleased to encounter the same in important non-safety systems such as reactor control systems. Except for examples needed to illustrate a point, this document is limited to predictable or partially predictable (a critical-task subset is guaranteed to meet deadlines) real-time systems. Unpredictable real-time systems impose such a burden of proof during safety determination that they are effectively ruled out in any safety system where consequences of failure are "unacceptable." Predictability as used here means that there are convincing arguments or mathematical methods that demonstrate that the system will meet its deadlines in all realistic situations. It is also assumed that any communication systems associated with systems considered here are determinate (see Preckshot 1993).

As will be developed in the section on timing (Section 3.2), systems of bounded indeterminacy (the frequency and duration of uncertain operations are bounded) are probably unavoidable, but are sufficiently predictable under certain restrictions. No real-time systems with unbounded-time computations are within the scope of this report. The report discusses mainly statically scheduled, state-based systems, with the proviso that dynamically scheduled, event-driven systems are admissible if convincing proofs of similar rigor can be made that deadlines will be met.

Fault-tolerant systems—for example redundant systems (Lala et al 1991)—are not covered, even though fault tolerance provisions often have impact on system timing. The position is taken here that fault timing can be accommodated by including fault hypotheses during the design process.

3. Technical Background

Guidelines for reviewing safety-critical real-time systems are not credible without technical support. This section provides the technical basis for the review guidelines proposed in Section 4. Since predictability is central to the thesis of this paper, some known methods of achieving it are discussed. Two subjects basic to deadline performance prediction, timing analysis and scheduling, are covered next. Then three areas important for implementing and testing real-time systems (prototyping, development approaches, and modeling and measurement) are covered. Lastly, there is a limited discussion of real-time operating systems that introduces some important features.

3.1 Predictable Real-Time Systems

Rather than try to answer the difficult general question, "What makes a system predictable?" this paper addresses the easier question, "What approaches have produced predictable systems?" There may be other ways of producing predictable real-time systems than those described here, but within the current art there is no general solution to the problem.

One way to demonstrate that a system will meet its deadlines is to show two things: that all tasks are schedulable, and that all tasks will be completed successfully before their deadlines if executed on schedule. A set of tasks is schedulable if, given that resource and priority conflicts are resolved, each task is completed prior to the time that it is scheduled for another execution, and CPU utilization (ratio of active CPU time to active plus idle time) is less than 1.0. For a dynamically scheduled system with unpredictable task phasing (task start times with relation to each other) and cycle times (period of task repetition), schedulability is a difficult thing to prove in general. In statically scheduled systems that do not stop, elementary reasoning suggests that tasks execute cyclically after an initial startup period (because the schedule is finite, but system operation time is unbounded). For such systems it can be shown that an independent task set (Ω: $\tau_i \in \Omega$) is schedulable by the rate-monotonic scheduling algorithm[1] if

$$C_1/T_1 + C_2/T_2 \ldots + C_n/T_n \leq U(n) = n(2^{1/n} - 1) \leq 1$$

where

n	is the number of tasks
C_i	is the execution time for task τ_i
T_i	is the period time for task τ_i
U(n)	is the least-upper-bound CPU utilization for n tasks.

(Sha and Goodenough 1990; Liu and Layland 1973)

The assumptions behind this expression are that task execution times are known, period times are known, and that several different period times (causing tasks to periodically vary phase between one another) can be accommodated in the same schedule. In other words, tasks are not required to have the same period. This still does not ensure that a non-conflicting schedule of dependent tasks can be attained (see rate monotonic scheduling in Section 3.3), but does indicate that task timing bounds and period times are a necessary condition for schedulability, and sufficient to guarantee schedulability of independent tasks. The rate monotonic schedulability relation above is optimal in the sense that no other fixed-priority scheduling algorithm can schedule a task set that the rate monotonic algorithm fails to schedule (Liu and Layland 1973).

If the deadline for task τ_i is D_i, then the deadline constraint offers two further bits of information. First, $D_i < T_i$, and second, $C_i < D_i$, where it is assumed that D_i is measured relative to the start of the task cycle.

For systems that do not directly satisfy the assumptions made above, certain regularity constraints may still allow predictability. Kopetz et al (1991) characterize hard-real-time systems based on real-time transactions, by which is meant a connected stimulus–calculation–response string of events. Such systems can be regularized if they can be restricted by a load hypothesis and a fault hypothesis. The load hypothesis defines a finite number of kinds of transactions, a non-zero minimum time between transactions of the same kind, and a non-zero minimum time between system recognition of stimuli. The fault hypothesis makes the same regular definitions of faults to which the system will respond. If the maximum computational time for each kind of transaction can be bounded, it can be shown that systems under regularity constraints have schedulability and deadline performance bounded by cyclic, rate-monotonic, fixed-priority systems of appropriately selected parameters.

Some dynamically scheduled systems can be made predictable for a subset of critical tasks by imposing regularity constraints that permit the reduced task set to meet the schedulability criteria above. The critical tasks must be able to preempt non-critical tasks, and the blocking time caused by resource

[1] A preemptive scheduling algorithm in which task priority is inversely proportional to task period.

conflicts with non-critical tasks must be included in the schedulability expression above (Sha et al 1990). This will be explained further in Section 3.3, Scheduling.

The criteria for predictability and schedulability as presented here are therefore:

- The system is periodic, and
- Task cycle periods are known, and
- Task execution times are bounded and known, and
- Deadlines are shorter than the task cycle, and
- Task runtime to completion is shorter than the deadline, and
- The task set satisfies the rate-monotonic least-upper-bound (LUB).

or:

- The system satisfies regularity constraints, and
- Bounding static system satisfies the rate-monotonic LUB.

or:

- A critical task subset satisfies regularity constraints, and
- The critical tasks preempt non-critical tasks, and
- Blocking time is included, and
- Bounding static system satisfies the rate-monotonic LUB.

Note that predictability does not imply feasibility. Resource conflicts may still make scheduling impossible.

3.2 Timing

Knowledge of timing behavior is central to predictability of real-time systems. The most basic timing knowledge describes the behavior of individual program units or hardware, and was called "module timing analysis" earlier in this report. This information is necessary to use schedulability expressions and scheduling algorithms, and is the first subject of this section. However, a single execution time is rarely defined for even simple segments of code, so some method for reasoning about execution times must be devised that includes timing variabilities. For hard real-time systems, the stochastic measures used in statistical performance analysis are not precise enough, so a way to speak of hard execution time bounds is needed. A method proposed by Shaw (1989) is discussed to illustrate the important issues. Some practical methods (software tools) for analyzing program unit execution time bounds are discussed, along with restrictions on unit software design. Finally, in distributed systems, separated parts of such systems do not perceive the same times, so that misorderings, anachronisms, and synchronization errors can occur unless measures are taken to prevent them. Some results in this area are introduced.

Hasse (1981) considered the execution of tasks synchronizing through critical sections (shared code that updates shared variables) in an attempt to estimate the time spent during these interactions. This work introduced the idea of "execution graphs," in which alternative sequences of access are enumerated and timing behavior is deduced for feasible orderings. Similar work on single programs used a "program graph" (Oldehoeft 1983) to enumerate alternative paths and simplify the expressions so obtained. Both of these papers demonstrated methods by which timing could be analyzed for either single or cooperating programs, and which appear to be the basis for later program unit timing analysis tools. Neither paper considered the effects of exceptions in detail.

Woodbury (1986) derived an expression for the probability distribution function for successful task completion time, including exception effects. The derivation includes two parts, one for exception

probabilities and one for task time distributions. Restrictions were placed on admissible tasks (later echoed or repeated in part by other workers). These restrictions were:

- Single entry/single exit.
- No preemption.
- Input/output is deterministic and each occurs at most once per task execution.
- Deadlines are met by abort if necessary.[2]

With these restrictions, a regular task structure graph was developed that permitted construction of analyzable real-time task graphs from elementary building blocks. By assigning *a priori* probabilities to branches and exception probabilities to individual instructions, Woodbury deduced probability distribution functions for active task time (F_{act}) and exception occurrence (F_{exc}). Combining these leads to:

Prob[T≤t, task done] =

$\{1 - F_{exc}(t)\}F_{act}(t)$ if $t < t_d$

$\{1 - F_{exc}(t_d)\}F_{act}(t_d)$ otherwise

where t_d is deadline time.

This work is significant because it demonstrates an analysis method that includes exception probabilities and provides a probabilistic estimate for successful task completion. Two useful results are that the analysis suggests ways to minimize $F_{exc}(t)$ and maximize $F_{act}(t)$. Since predictable systems need virtual certainty about task timing, a probability distribution of task completion times that approaches 1.0 before t_d is quite useful.

While the work of Hasse, Oldehoeft, and Woodbury showed that there were no logical barriers to task timing predictability, one very important factor, hardware, was not dealt with in detail. Park and Shaw (1991) have found that some hardware with dynamic memory refresh may exhibit as much as 7% excess random delay unaccounted for by instruction time addition. In another report (Callison and Shaw 1991), as much as 15% random delay due to memory refresh and bus synchronization during interrupt service was found. This is disturbing but not unmanageable if the indeterminacy is bounded. "Interferences" of the types reported can be handled if the frequency and duration of the interference are bounded. An example given for a clock interrupt illustrates the method. An obvious extension permits calculation of minimum and maximum bounds on execution time.

$$t^m_p = t'_p \left(\frac{p_{clock}}{p_{clock} - t_{clock}} \right)$$

where

t^m_p is the measured (actual) program execution time

t'_p is the pure (calculated) program execution time

p_{clock} is the clock period time

t_{clock} is the clock interrupt service time.

The reports by Shaw and others suggest that no realistic modern computer system is exactly predictable, but that bounds on minimum and maximum task execution times can be calculated. Shaw (1989) proposed a formal method of reasoning about such timing behavior. In precis, since earlier tasks have uncertain ending times, the starting time of a new task τ may be uncertain but in a bounded interval

---

2 A technique mentioned in Section 3.3, Scheduling, called "imprecise computation" replaces failed (overtime) calculations with a quick approximation, allowing deadlines to be met.

(a,b). Because τ also contains some uncertainty, it finishes in a bounded interval (c,d) that is not necessarily a simple offset of (a,b). Thus, given a sequence Σ of tasks that starts in (p,q), an algebra of intervals is required both to determine the ending interval of Σ or the starting and ending intervals of any $\tau_i \in \Sigma$. Timing tools for real systems must use some variant of this reasoning if accurate results are expected. The utility of lower bounds on task execution times becomes immediately obvious, since both upper and lower bounds are required for interval arithmetic.

Timing analysis tools that use the principles just mentioned have been devised and demonstrate the feasibility of performing the analysis. Park and Shaw (1991) have built a timing analysis tool which analyzes a subset of the C programming language. The tool uses a compiler front-end to generate a syntactical analysis of program structure, which is then characterized in terms of "atomic blocks," or compiler-generated machine code sequences of known timing behavior. The block structure was then combined under time reasoning rules to produce lower and upper execution time bounds for code segments or programs. A limited number of tests were performed for small and large atomic block granularities, with generally successful predictions.

A considerably more advanced program timing tool that is part of the Mars[3] real-time operating system development environment (Pospischil et al 1992) places restrictions on allowable tasks that are reminiscent of Woodbury (1986). These are:

- Input at beginning.
- Output at end.
- No internal communication.
- No gotos.
- No recursions.
- All loops statically bounded.
- No dynamic memory allocation.

These limitations allow the timing tool to compute program time bounds from information supplied directly from the code. To make the timing bounds tighter, this timing tool employs programs that supply auxiliary information to permit more accurate calculations:

- Scope—code block with one entry, one exit.
- Marker—limit on executions of a scope.
- Loop sequence—interdependencies of loop bounds.

The Mars timing tool is embedded in a development environment that includes two editors, a compiler, and two off-line schedulers (Kopetz et al 1989; Pospischil et al 1992). These components are used in the iterative sequence:

- Text editor accepts programmer entry.
- Compiler generates timing tree structure.
- Timing editor allows speculative times for unknowns.
- MaxT[4] analysis inserts calculated times for knowns.
- Task and bus schedulers generate schedules.

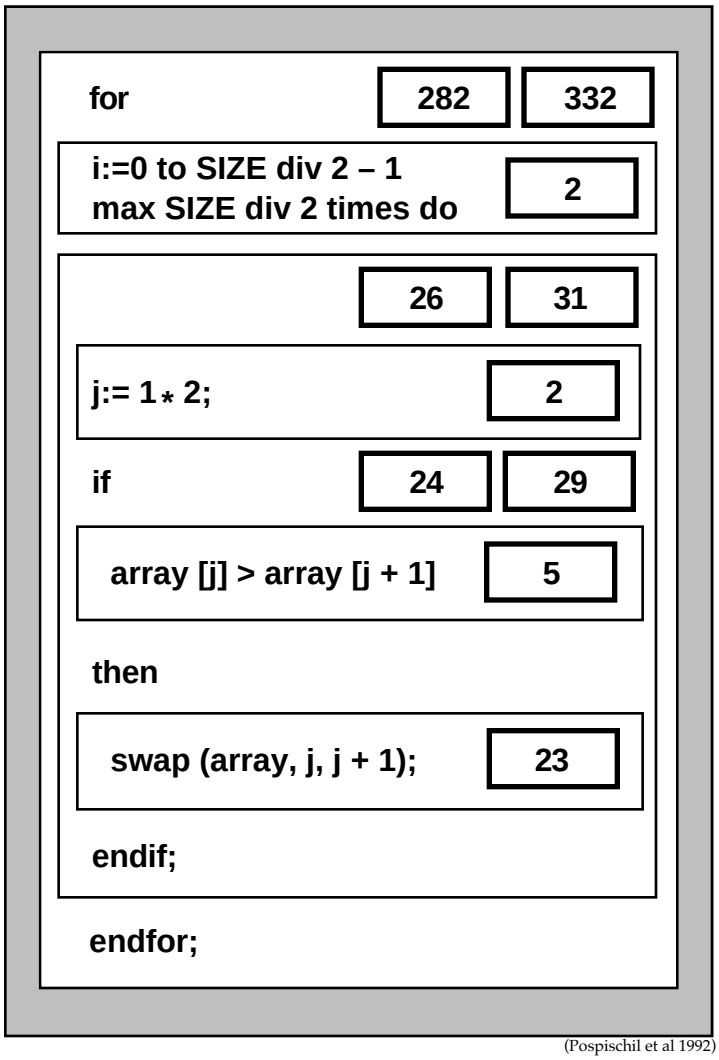
Because of the graphical presentation of the editors (see Figure 1), the programmer is able to iteratively refine real-time task performance so that a feasible schedule (all tasks are scheduled and all deadlines are met) can be attained.

---

[3] Mars (MAintainable Real-time System) is a research real-time system developed at Technische Universitat Wien (Vienna), specifically for investigating predictability of real-time systems.

[4] The maximum time a particular computation is expected to take.

The Mars and the Park and Shaw timing tools are examples of automatically generated program timing information. It is also possible, but tedious, to prepare this information by manual analysis (Smith 1990).



(Pospischil et al 1992)

Figure 1. Mars Timing Editor Presentation.

In distributed systems, knowledge of local program time is insufficient to characterize overall behavior of the system and to schedule node-to-node communications relative to individual node activity. This is because clock skew between system nodes is unavoidable, and time can only be locally ordered[5] (Lamport 1978). This broader view of interacting modules was called "system timing analysis" earlier in this paper. Clock skew and partial time ordering cause confusion between nodes (if no corrective measures are taken) of the following types:

- Absolute times of event.
- Causal ordering of events.
- Calculation of time intervals.
- Data consistency (timestamps).
- Uncoordinated scheduling of tasks on separate processors.

Lamport suggested a method of ordering "events" on a global scale by exchange of timestamps in messages so that there was a relation to physical time within an error bound, even though individual computer crystal clocks drifted relative to each other. The method involves using timestamps in messages that are more or less regularly exchanged to make positive corrections to local clocks. Negative corrections are not allowed, since this would make local clocks non-monotonic in some instances. Later work by Kopetz and Ochsenreiter (1987) uses a network interface with an embedded timestamper (timestamps are automatically added to outgoing and incoming messages) to maintain clock synchronization between physically separated nodes on a network. An algorithm is proposed for calculating clock corrections that the authors maintain is good to about 100 μsec while utilizing about 1% of CPU time. Both of these methods will ensure that timestamps from anywhere in the system are within some small error of the correct universal (e.g., astronomical) time, but they may impose an order on concurrent events within the error window that is unrelated to causality (Fidge 1991). In some real-time systems this may not be a problem, because causality is assumed by other logic, or a consistent total ordering is all that is necessary for resource allocation algorithms or deterministic communication scheduling.

In systems where exact causal ordering is required, Fidge (1991) has proposed "logical clocks" by which causal ordering or concurrency[6] can be exactly detected. The Fidge method uses a "vector" clock representation of time by which communicating tasks retain a vector of last-known local clock times of all tasks with which they interact, either directly or indirectly. Updates occur at message interchanges. By comparing other tasks' ideas of time, a task can make decisions about event orderings in remote parts of the system. Logical clocks maintain an order consistent with causality, but not necessarily precisely in step with astronomical time. Some time intervals may contracted or extended.

A global theory of time in distributed systems is important for the previously mentioned scheduling purposes, and also for ensuring that synchronization occurs properly between cooperating portions of the system. This is generally the main concern in communication protocol design (Preckshot 1993), and some communication protocol techniques for validating specifications are used in real-time system analysis. In particular, reachability analysis has been combined with symbolic execution (to eliminate infeasible states) to explore such synchronization problems as deadlock, livelock, unspecified messages, and state transition difficulties (Young and Taylor 1988).

---

5 Resulting in system-wide partial ordering.

6 Two concurrently executing tasks detect events but do not exchange messages. It is impossible, in the absence of message interchange, to determine event ordering except that the events occurred in the same interval.

Summarizing real-time system timing issues: with suitable restrictions, real-time code timing can be predicted within deterministic bounds; computer hardware exhibits bounded indeterminacy; there are methods for logically and physically synchronizing distributed systems within a small time error.

3.3 Scheduling

The final kind of timing analysis, schedulability and priority analysis, has to do with *when* and *in what order* tasks are executed. Once code timing is known, or (with something like the Mars timing editor) code timing estimates are known, a tentative schedule can be constructed if static scheduling is being used. Static scheduling has the very distinct advantages that the method used to calculate the schedule is not a significant factor during runtime, and the schedule (or schedules) can be validated before use. Some dynamic scheduling algorithms may themselves use substantial CPU resources, and so contribute to the problem they attempt to solve.

If it were just a matter of concatenating code runtimes and checking that sufficient CPU cycles were available, scheduling would be easy. Unfortunately, if real-time code is at all complex, some code talks to other code or uses the same resources. At the very least in distributed systems, the use of interconnecting communications media must be arbitrated. In predictable systems, this means that it must be possible to predict that all necessary messages will be delivered in time to allow recipient code to meet deadlines or, more restrictively, to determine what messages will be sent and when they will be sent. The most definitive and direct course is to lay out task runtimes, resource usage intervals, and communication channel usage, and demonstrate that there are no conflicts. This is the approach taken in static scheduling, and is exemplified by the Mars scheduling approach (Kopetz et al 1989). The Mars system permits several previously calculated schedules to be used one at a time, depending upon system needs.

Since the scheduling algorithm used for statically scheduled systems is immaterial, provided it produces workable previously calculated schedules, algorithms reported in use in dynamically scheduled systems will be discussed without further mention of static systems. As previously mentioned, the rate monotonic algorithm is optimal among fixed-priority scheduling algorithms, and produces a least upper bound on processor utilization of:

$$U(n) = n(2^{1/n} - 1)$$ where n is the number of tasks (Liu and Layland 1973).

Other algorithms are in use besides the rate monotonic algorithm. Zhao et al (1987a) state that the general optimal scheduling problem (scheduling randomly arriving tasks with indeterminate computation times, deadline and resource constraints) is "NP-hard" (time to solve increases faster than any finite power of n, the number of tasks). Consequently, Zhao proposes a heuristic that produced a satisfactory but not optimal schedule in several examples given in the paper.[7] It is instructive to use this as an example of techniques used in scheduling. Zhao represents the task set Ω composed of tasks τ_j to be scheduled as characterized by:

1. Processing time C_j
2. Deadline D_j
3. Resource requirements R_j, where $R_j = \{R(p), R(q),...R(z)\}$
4. $R_j(i) \in R$, a set of r resources
5. $1 \le p \le q \le z \le r$ are indices in R associated with τ_j.

[7] It cannot be proved that a heuristic algorithm will always be correct, but in statically scheduled systems it is only necessary to prove that the resulting schedule is correct.

The set of non-preemptable tasks is scheduled on a series of time slices S_k which occur at task completions (or in later work, also at preemptions (Zhao et al 1987b)). Time slices are similar to the familiar time slice of time-sharing operating systems, except that they do not occur except at exceptions, completions, or preemptions. A schedule is full if for all j, the task τ_j processing time C_j is less than the sum of all time slices in which τ_j executes. A partial schedule includes tasks which have insufficient slice time to complete. A schedule is fully feasible if for all j, the schedule is full and the end of the last slice time S_k for each task τ_j is less than D_j. A fully feasible schedule is built by starting with a null schedule and extending it one task at a time. Each extension is partial until a full schedule is reached. The first task is selected using smallest-laxity-first (laxity is deadline minus current time minus computation time), after which the heuristic algorithm is used to select additional tasks. The authors define conditions called strong- and weak-feasibility, with a proof that weakly feasible[8] schedules cannot be extended to full feasibility, resulting in a search cut-off heuristic. They provide a formula for determining strong feasibility of partial schedules.

The Zhao technique illustrates two important points about dynamic scheduling algorithms. The first is that when task τ_j arrives, there may already be a schedule in place, so the algorithm must provide some method of extending an existing schedule. The second is the dilemma posed by optimal versus heuristic algorithms. An optimal algorithm may in theory be able to schedule a task set, but not in time to do any good. A heuristic algorithm may work quickly enough, but may fail to find a feasible schedule.

The rate monotonic algorithm has been extended to systems with periodic and aperiodic tasks with priority inversion (Sha and Goodenough 1990; Sha et al 1990; Warren 1991). In simple rate-monotonic scheduling, the priority of periodic task τ_j is inversely dependent upon its period T_j, and higher-priority tasks preempt lower-priority tasks. This method works for independent tasks, but independence is rarely the case in distributed, communicating real-time systems. Tasks interact by communication and resource conflicts. In preemptive systems, a condition known as priority inversion occurs when a lower-priority task blocks a higher-priority preemptive task (and thus deadlocks) by possessing a resource the higher-priority task needs when preemption is attempted. The situation is resolved by having the lower-priority task inherit the priority of the preempting task while it is in the critical section of code (Sha et al 1990). Schedulability calculations are extended by adding the appropriate blocking time to the schedulability relation as:

$$C_1/T_1 + C_2/T_2 \ldots + C_n/T_n + \max(B_1/T_1, B_2/T_2, \ldots, B_{n-1}/T_{n-1}) \le U(n) = n(2^{1/n} - 1)$$

where:

n	is the number of tasks
C_i	is the execution time for task τ_i
B_i	is the longest blocking time for task τ_i[9]
T_i	is the period time for task τ_i
U(n)	is the least-upper-bound CPU utilization for n tasks and priority is ordered highest = 1 to lowest = n.

[8] Weak feasibility is a mathematical condition of an incomplete schedule and tasks remaining to be scheduled that indicates that no further tasks may be added without violating deadline constraints. The details are beyond the scope of this report.

[9] From a system response standpoint, blocking times should be kept as short as possible, even though the schedulability relation is satisfied.

Sha and Goodenough present an additional mathematical test (not repeated here) for determining if a periodic task set will meet its deadlines for all possible task phasings. A problem remains that aperiodic tasks are not included in the theory. The theory is extended to high-priority aperiodic tasks of bounded frequency and execution time by scheduling an "aperiodic server" that is, in fact, a fictitious periodic task with period equal to the frequency bound of the aperiodic task. Provided that the computation time of the aperiodic task is well within the CPU allocation for the aperiodic server, random preemptions whose frequency does not exceed the frequency bound will be serviced rapidly.

The "Spring Kernel"[10] (Stankovic and Ramamritham 1991) is an approach that uses a combination of dynamic and static scheduling. Tasks are divided into critical, essential, and non-essential tasks, with critical task deadlines being guaranteed by previously calculated static schedules. Essential tasks are dynamically scheduled by an on-line scheduler that attempts to guarantee deadlines by using auxiliary information not usually available to general-purpose operating system schedulers. Stankovic and Ramamritham suggest that because real-time systems are more tightly constrained than general-purpose computing systems, more is known about tasks in the system and should be used to advantage by the scheduler. The kinds of auxiliary information they propose are:

- An equation for worst case execution time.
- Deadlines, periods, or other real-time constraints.
- Priority.
- Resources needed.
- Importance.
- Whether the task is "incremental."
- Precedence graph (of interacting tasks).
- Communication graph (of communicating tasks).

An incremental task is a special case of an imprecise calculation. An imprecise calculation is a less-accurate or heuristic calculation that is used to produce some sort of result when a more-precise but run-length unpredictable algorithm fails to produce timely results. Imprecise calculations are used to ensure that task calculation times are bounded. Incremental tasks produce imprecise results early in the process, but continue to refine the results until done, or a calculation deadline is exceeded. In the Spring Kernel approach, the combination of numerical expressions for task run time and imprecise calculation information give the scheduler a better chance to construct a feasible schedule.

The important points about scheduling are worth recapitulating:

- The scheduling algorithm used in statically scheduled systems is immaterial, provided it produces workable schedules.
- The most definitive and direct course is to lay out task runtimes, resource usage intervals, and communication channel usage, and demonstrate that there are no conflicts.
- $U(n) = n(2^{1/n} - 1)$ is a conservative least upper bound of CPU utilization for a set of n tasks.
- A dynamic scheduling algorithm must provide some method of extending an existing schedule.
- An optimal algorithm may, in theory, be able to schedule a task set, but not in time to do any good.
- A heuristic algorithm may work quickly enough, but may fail to find a feasible schedule.

---

[10] The Spring Kernel is a research real-time operating system developed at the University of Massachusetts at Amherst to investigate predictability and performance attributes of large, complex real-time systems.

- The rate monotonic algorithm has been extended to systems with periodic and aperiodic tasks with priority inversion.
- More is known about tasks in real-time systems than in general time-sharing systems, and can be used to advantage by dynamic schedulers.
- Imprecise calculations can be used to ensure that task calculation times are bounded.

3.4 Prototyping

The issue of what prototypes to build and when to build them is complicated by the variety of purposes that prototypes serve. For convenience, an expanded form of earlier discussion of prototype kinds (Taylor and Standish 1982) is presented here in list form:

1. Feasibility prototype (do not know how to build it).
   1a. Abstract requirements but too difficult in practice.
   1b. Correct requirements but do not understand environment.
2. Requirements prototype (do not know what to build).
3. First of type off line.

The danger of all prototypes is that, driven by economic pressures, the prototype becomes the product. This is particularly true of software prototypes—they tend to be thrown together in a hurry, like temporary military buildings, and then amaze everyone by a rickety survival ten times longer than anyone expected. There is therefore some hazard in requiring prototypes without carefully prescribed purpose or limits to usage.

Kind 2 software prototypes are usually built either to get user feedback on how a prospective system should function, or to try several known approaches to a problem to see which suggests the most cost-effective solution. The most likely usage in safety-critical embedded systems (such as reactor protection systems) is for human-factors investigation of surveillance difficulty and maintainability. Safety-related annunciation systems can also be reviewed in prototype form to ensure that operators are not confused by reports from the display system during stressful situations. In the former case, an actual physical prototype with minimal software would be needed, since surveillance and maintenance access requires manual interaction. In the latter case, artwork mockups and hand-replacement of imitation displays might be sufficient. Since there is substantially more user interaction with plant control and process parameter display systems, Kind 2 prototypes for these systems might be considerably more comprehensive. For reactor protection and reactor control systems, it is probably advantageous to consult human factors specialists to determine the extent of Kind 2 prototypes required.

Kind 1a prototypes are built because, while system requirements may be exact, they are sometimes too abstract to engender a practical response. An example would be, “design an automobile-mounted system that avoids automotive collisions.” Everyone understands exactly what is wanted, but nobody can build it because they do not know how. The situation pertains in part to reactor protection systems, because the reliability requirements for software may be unattainable in general, or a new reactor design may introduce new physical behavior that protection system software must track. In the first case, constructing a software prototype to achieve modest gains in reliability may be useful, but where requirements are significantly beyond the limits of practice, adjusting requirements to more realistic levels may be the only reasonable course. In the second case (a core protection calculator for a new kind of core, for example), the requirement is known (trip the reactor if the core approaches safety limits), but the prototype is necessary to work out the protection algorithm details.

Kind 1b prototypes probably will be useful in all real-time computer systems associated with nuclear reactors. A case has been described (bounded indeterminacy in computer systems (Park and Shaw 1991)) in which measurement of the hardware environment provides significant information for predicting performance. Additional areas include communication system indeterminacy, sensor readout random variations, and computer interface characteristics. Kind 1b prototypes usually involve hardware and test software for collecting data needed early in the design and performance prediction process. For this reason, they are more useful early in the design process rather than later, when the news may be too late for economic correction.

The final kind of prototype in the above list, Kind 3, is typically used for validating that the production process has produced something that meets the original specifications. For computer-based protection systems and control systems, these prototypes would generally be sufficient portions of a final system for software–hardware integration and validation testing. For ultra-reliable systems, there is some sentiment that the validation prototype must be the final system, since no change in software can be tolerated without fatally disturbing the validation process. This may not be the case for performance issues, however, so this paper leaves ultra-reliability issues to Lawrence (1992) and Barter and Zucconi (1992). The performance objectives of final prototypes are to show that:

- The performance models used match the product.
- Performance scales as predicted.
- Unforeseen performance "glitches" are unlikely.
- Performance equals or exceeds requirements.

Because models play such a significant role in real-time system development, decisions about final prototypes necessarily should be guided by model development and predictions. The model is often a useful guide to how much of the final system must actually be prototyped to resolve outstanding performance questions. These issues will be covered in more detail in Section 3.6, Modeling and Measurement.

3.5 Development Approaches

Development of real-time systems was originally an ad hoc process and, unfortunately, it still is for some developers. Tools used for development include the "In-Circuit Emulator" (ICE),[11] that allows the software engineer to follow code execution at the machine language level. There are software tools to support ICEs that allow the engineer to follow code execution at source language level, but there is still a regrettable tendency to resort to assembly language much too often. This paper will not further consider such approaches since the risks involved exceed what is tolerable for reliable reactor protection systems. The use of ICEs is not ruled out, but should be limited to specific performance problems where especially tight timing must be demonstrated, or where performance is being verified. In other words, high-level language based development method, timing analyses, and proof of schedule correctness are preferred over code development at the machine level.

3.5.1 Tuning Performance During Integration

A systematic approach that was popular in the 1980s, and still is in some quarters, can be described as "make it right, then make it go fast." Developers use some software development method that helps to produce reliable code, and then discover code execution times by experiment. A software tool called a

[11] The ICE is a very commonly used electronic tool available from computer system vendors and independent equipment suppliers.

"profiler" is used to determine which modules are getting most use ("hot spots") and these modules are rewritten if necessary to speed up system execution. ICEs may be used in place of profilers in embedded systems that have limited accessibility.

This method works for systems of limited extent, but can run into trouble if performance problems are dominated by poor system architecture rather than isolated code module performance. It also has the defect that performance problems are not discovered until late in development, when little time (and money) remains to correct them.

3.5.2 Engineering Performance During Design

More recent methods include specifying performance requirements early in the design so that appropriate system architectures can be chosen and performance requirements can be allocated between parts of the architecture. This is not an easy thing to do *a priori*, so most such methods include iterative modeling techniques to work out design alternatives.

An example of this approach is provided by Smith (1990), who calls her methodology Software Performance Engineering (SPE). SPE has not been used for hard real-time systems,[12] but a description illustrates the iterative modeling and measurement sequence well. Initially, requirements are captured without constraints. Requirements analysis confirms whether or not the requirements are feasible, and suitable designs are identified. In a pattern that will be repeated throughout the design cycle, best- and worst-case scenarios are developed. Early performance models are devised and exercised to predict best- and worst-case performance, and the design is modified as necessary to bring worst-case performance within requirements. As design progresses through implementation, more and more of the model(s) is (are) replaced by data obtained from actual performance measurements, and both the model(s) and the design are iteratively corrected to match actual performance and to meet performance goals.

The methodology described by Kopetz (Kopetz et al 1991) used in the Mars development environment follows a similar scheme, except that deterministic rather than statistical timing constraints are used. The Kopetz method models real-time systems as a set of real-time transactions (a stimulus–computation–response event string) that is subject to deadlines. Transactions are broken down to tasks and distributed to physical nodes, which results in a set of performance requirements for both tasks and communications between tasks. Using the previously mentioned text and timing editors (Pospischil et al 1992) (Section 3.2, Timing) and static scheduling software (Section 3.3, Scheduling), an iterative process of refinement is followed until a set of predictable tasks and a feasible static schedule have been implemented.

In both approaches just mentioned, the methods reach the validation stage with well-developed performance models, which are aids to choosing the appropriate late-stage prototype and devising performance tests. From a performance validation standpoint, performance models are quite helpful because they afford more insight into the system being scrutinized.

3.5.3 Early Prototyping

Another school of thought proposes early or "operational" prototyping (Davis 1992). This is actually a staged or "versioned" stepwise development approach in which progressively more capable systems are developed and released as "versions" or "major releases." Since early versions have little capability, it is usually easy to achieve performance goals. The stepwise approach distributes the effort and risk of meeting performance objectives over a longer time, during which more can be learned about the problem, the hardware, and the environment. The method is favored in situations where there are substantial

---

[12] The usual application domain is transaction-type soft real-time systems where statistical throughput measures are acceptable.

unknowns, insufficient money for full system development, and vague requirements. It is the de facto method for much commercial software, real-time or not. Early versions have the reputation of being quite "buggy." Staged implementation may not be a good choice for reactor protection systems because substantial unknowns and vague requirements are not tolerable in safety-critical systems. Commercial software products that use this development approach should be used with extreme caution in safety applications.

3.5.4 Special Languages

Specially designed computer languages have been proposed to make the development of real-time systems easier or more accurate. Some of the more extravagant claims suggest that correct language design will make it difficult to write incorrect code. To date, that has been the ideal rather than practice. Real-time computer languages can be split roughly into two subgroups: specification languages and implementation languages. Some examples of the former are Task Sequencing Language (TSL) (Rosenblum 1991), Extended Temporal Description Language (TPDL*) (Cabodi et al 1991), RT-ASLAN (Auernheimer and Kemmerer 1986), and various Formal Description Techniques (FDTs) used in communication protocol specification (Preckshot 1993). Examples of the latter are Occam 2 (Burns and Wellings 1990), Modula II (Burns and Wellings 1990), Flex (Kenny and Lin 1991), and the U.S. Department of Defense Ada (Leveson et al 1991; Burns and Wellings 1990; Clapp et al 1986).

Specification languages attempt to provide an unambiguous way to describe timing and sequencing specifications. Some of them have been translated into models that can be utilized to perform reachability analysis (Preckshot 1993) and thus eliminate certain synchronization hazards. It is still necessary, once a system has been described in a specification language, to convert the specification language form to an executable language form. The practical effect of these specification languages on performance is unclear, although having an unambiguous timing requirement must be counted a plus.

Real-time implementation languages usually try to provide parallelism and synchronization constructs. Burns and Wellings (1990) list three general areas:

- Concurrent execution through the notion of processes.
- Process synchronization.
- Inter-process communication.

Other constructions include monitors (Hoare 1974), mutual exclusion markers (Holt 1983), explicit parallelism, and rendezvous mechanisms such as that used in Ada. Occam 2 is unusual in that it assumes language statements can be executed in parallel unless specifically told they must be executed in sequence. The implementation languages are more concerned with getting concurrent program interactions correct than with optimizing performance. In this respect they support the performance tuning approach at integration, rather than building performance in at design time. There is often the implicit assumption that more performance is available through greater parallelism, but explicit support for deadlines is not generally the case. Ada and Flex support exception mechanisms that allow imprecise calculations to be substituted for tasks that are close to exceeding a deadline. Flex has some characteristics of a specification language because code blocks are enclosed in "constraints," which act like timing specifications. The exception approach still requires assistance from an operating system or runtime kernel.

3.6 Modeling and Measurement

Modeling is a necessary discipline when building real-time systems. Models serve as performance predictors during initial development stages, and later as guides for validating the system. One of the most significant models in use, the static schedule, may not be recognized as a model because, like the "purloined letter," it is too obvious. Static schedules are exact models for how a statically scheduled real-time system is supposed to perform, and can be used directly to verify that software execution and communication occur precisely as planned.

The other models generally in use for real-time system development are code execution models, system-level Petri net models, and queuing network models. Code execution models were previously discussed in Section 3.2, and they produce code timing data that is used by the system-level models. This section will discuss variants of the other two system model types and methods of measuring software performance for comparison to model data.

Petri nets (Peterson 1977) are a method of representing systems that go through complex but finite state transitions. They are usually represented graphically as a directed graph of three types of objects (places, transitions, and directed arcs) and a "marking" that describes the system state. Figure 2 shows such a net, with the marking being the solid dots, which are called "tokens." Mathematically, the simple Petri net is stated:

$$PN = (P, T, I(t), O(t), M)$$

where

P	is a set of places,
T	is a set of transitions,
$I(t_j)$	is the set of input places for transition t_j,
$O(t_j)$	is the set of output places for transition t_j,
M	is the marking.

The model is executed by firing transitions, which results in a new marking. The sequence of markings obtained by transition firings represents a trajectory through the state space of the system being modeled. In the simple Petri net, transitions may fire as soon as they are enabled (at least one token is in each input place of the firing transition). The new marking is determined by removing one token from all input places of the fired transition and placing a token in all output places of the fired transition. The reachability set of a Petri net is the set of all markings that can be reached by execution from an initial marking M_0. Note that because there may be ambiguities about which transition will fire when two transitions share the same input place (a "conflict"), determining the reachability set may be non-trivial or indeterminate. Finding the reachability set is called reachability analysis and is equivalent to finding the states that a correctly modeled system can assume. Ambiguities, non-conservation of tokens, more than one token in an input place, dead-end markings, and unexpected markings are all findings that may indicate logical problems with the original system but may also be modeling errors.

The original definition of the Petri net included no concept of time. Petri nets have been extended to timed Petri nets, Stochastic Petri Nets (SPNs), and Generalized Stochastic Petri Nets (GSPNs) (Conte and Caselli 1991). GSPNs illustrate the method of extension quite well. The mathematical description of the GSPN is:

$$GSPN = (P, T, \Pi(t), I(t), O(t), H(t), W(t), M),$$

where

P	is a set of places,

T is a set of transitions,
$\Pi(t_j)$ is the priority of transition t_j,
$I(t_j)$ is the set of input places for transition t_j,
$O(t_j)$ is the set of output places for transition t_j,
$H(t_j)$ is the set of inhibitory places for transition t_j,
$W(t_j)$ is the rate or weight of transition t_j,
M is the marking.

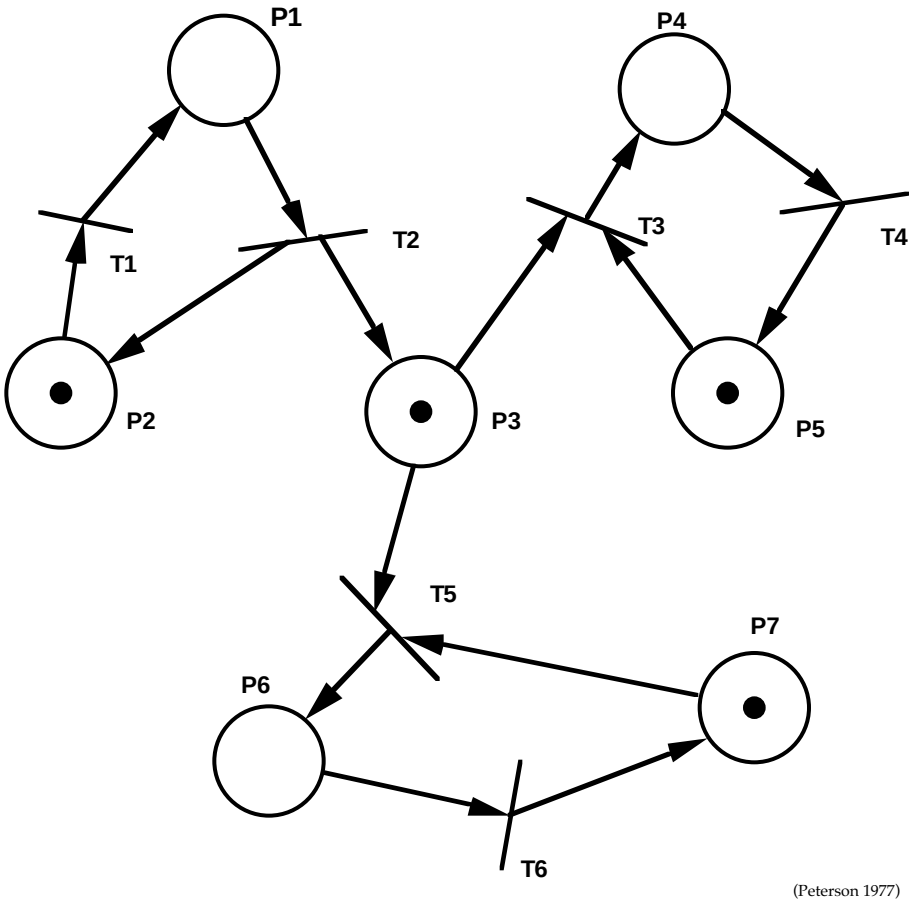


Figure 2. A Petri net Showing Conflict Between Transitions T3 and T5.

Two new features for arbitration of conflicts, Π(t) the transition priority, and W(t) the rate or weight, are added. W(t) can be a probability distribution function for firing delay, a simple firing delay, or a probabilistic weight to be used to select which of several enabled but conflicting transitions will fire. H(t) now allows modeling of inhibitory behavior (a token prevents firing) as well as permissive behavior. The augmented Petri net models can model timed behavior, but still do not directly model queuing behavior, which is often explicitly coded into real-time system designs. Petri nets may also be executed directly (as a simulation) or reachability graphs can be converted to a continuous-time Markov chain model, from which statistical performance indices are then computed.

Queuing network models (Lazowska et al 1984) were among the first models to be used to represent computer systems that provided services to a user program population that was known only statistically. Figure 3 shows a simple queuing model of a single CPU and four disks that can be used in parallel (MacDougall 1987). Associated with each queue is a service time (usually modeled probabilistically) and, where multiple output paths are possible, a probability for selection of each path. Work coming into the system is described as a stochastic generation process; system throughput

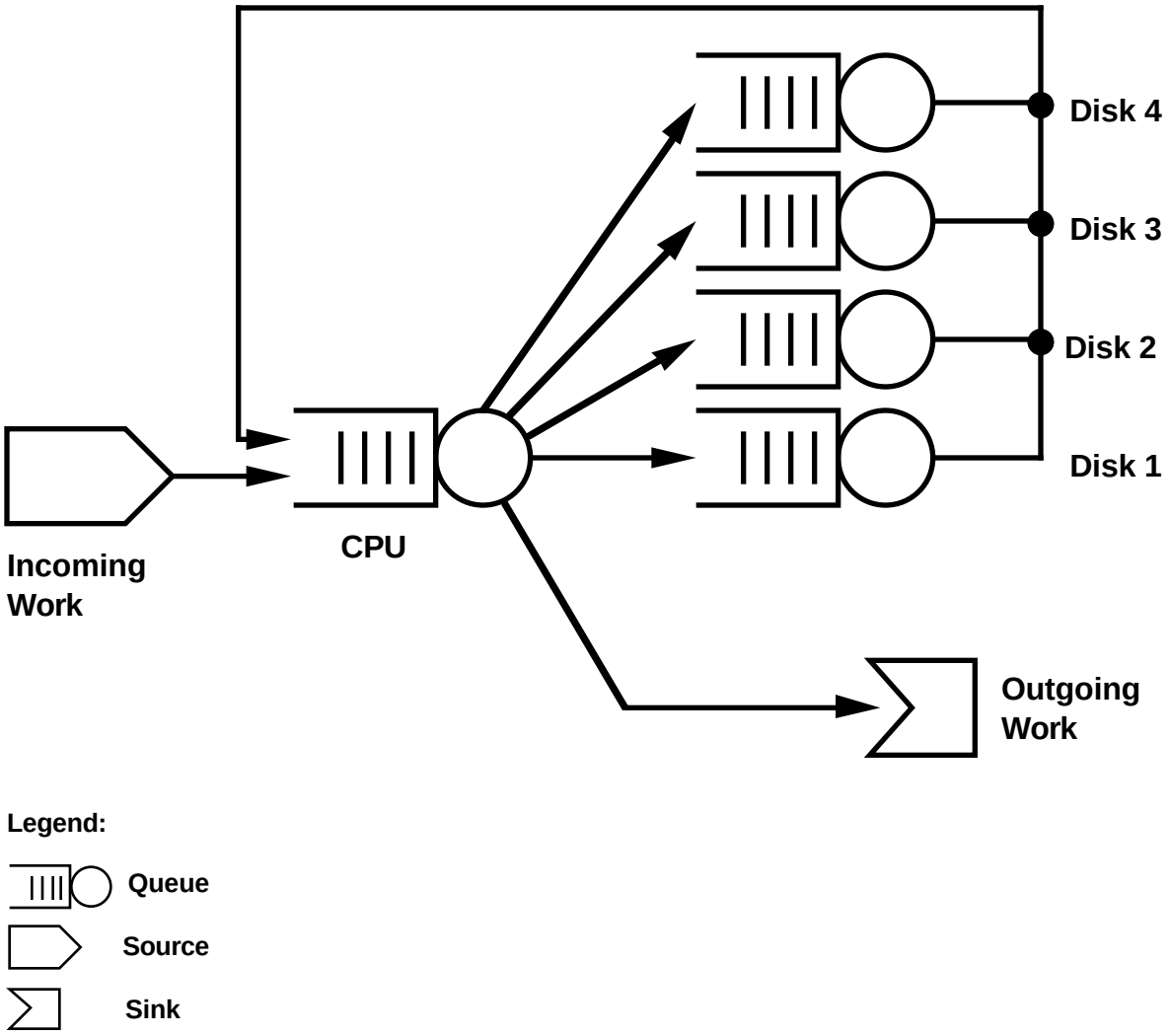


Figure 3. A Simplified Queuing Network Model of a Computer System with Four Disks.

response, including queue loadings, are some model outputs. Like Petri nets, queuing network models can be used for simulation or solved analytically. They are often used (Smith 1990) for performance analysis of systems for which average or expected throughput under specified load conditions is desired. There are two defects when used for analysis of hard real-time systems: average performance is not acceptable when there are hard deadlines, and model output is very dependent upon accurate workload characterization. In spite of this, these models may still be useful to understand system operation if deadlines are guaranteed by other logic. In particular, performance predictions may serve both as initial feasibility confirmation and as later testing yardsticks.

An obvious question is, "Why not combine Petri nets and queuing network models?" The Timed PQ Net (Chang et al 1989) is such a combined model. Queues have been added to transitions, which now obscure the location of marking tokens.[13] No significant usage has been reported with this model yet.

Once code has begun to be implemented, continuous comparison with model predictions is necessary; otherwise there very likely will be a steadily widening gulf between planned performance and actual performance. Measurement techniques are needed during development and during integration to demonstrate performance to developers and regulators. From Smith (1990), measurement is usually performed in four situations:

1. Measurement data from existing systems.
2. Measurements of experiments or prototypes.
3. Measurement of actual code units on different hardware and then extrapolating to the target system.
4. Measurement of the target system.

The measurement situation often governs which measurement tools are possible or available. The tools generally available are:

- System monitors (e.g., task resource and time usage).[14]
- Program monitors (profilers).
- System event recorders.
- External measurement devices (logic analyzers, ICEs).
- Instrumented programs.
- Benchmarks.

System monitors and program monitors are tools that need support from some sort of operating system. The difference between the two is that program monitors provide more detailed information about the internals of program operation, and often need access to source code and a syntactical structure chart. System event recorders can be software or hardware devices that note the time of predefined "events." Loggers would fit this definition, and are often required on real-time control systems to maintain a record of "significant" actions. All three of these measurement methods are better suited to established systems or systems with substantial operating system support (situations 1, 3, and 4 above).

External measurement devices have the advantage that they generally do not interfere with running code, but merely observe it. This is balanced by the difficulty of application, since invasive hardware procedures are required, and it is hard to tie the machine language instruction stream to source code without coupling with the compiler and linker. In spite of these difficulties, external measurement is often used in situations 2 and 4 above because no acceptable alternatives exist.

---

[13] Tokens may now exist in a queue in a transition or in a place, which violates the initial precept of Petri that places contain the tokens.

[14] These are not monitors as proposed by C.A.R. Hoare (1974).

Instrumented programs, which are programs with extra performance measurement code compiled into them, are used in all situations and in combination with other measurement techniques. The simplest form of instrumentation is loop counting or clock watching, and it is very effective. Probably the highest frequency of use for instrumented programs is in unit performance testing. Execution time of a code module can be easily measured by reading the system clock, executing the module N times, and rereading the system clock. Module time is then the elapsed time divided by N. A special form of instrumented program is called a "benchmark," which is a calculation performed N times to measure processor speed for particular kinds of computations. Much has been written about benchmarks, mostly obscuring their real effects because of advertising hyperbole. Regardless, benchmarks are useful point measurements when specific types of calculations are important to real-time performance.

3.7 Real-Time Operating Systems

Real-time computer systems are usually designed in one of two ways. The first way is to design the entire system from scratch, writing all of the code that ends up in the target computer system as a dedicated application or coordinated group of applications. The second way is to separate the applications from commonly used "system services" and either to write or acquire a "real-time operating system" to provide these system services. Real-time operating systems are also called "embedded kernels" or "real-time kernels," and as the names suggest, are the core around which the applications are built. It is very likely that real-time operating systems will be encountered sooner or later in nuclear power plant computer systems. This section will discuss the general features of some commercially offered real-time operating systems, as well as two research systems reported in the literature.

Because of the number and variety of offerings calling themselves "real-time operating systems," it is beyond the scope of this work to review even a portion of them. To name several, there are VxWorks and the Wind River Kernel (Wind River Systems, Inc.), OS-9 and OS-9000 (Microware Systems Corporation), VMS and VAX ELN (Digital Equipment Corporation), VRTX (Ready Systems), LynxOS (Lynx Real-Time Systems, Inc.), and UniFLEX (RTMX-UniFLEX). These systems share some generic features, among which the most consistent seems to be "configurability." In order to satisfy the largest possible number of potential customers, real-time operating system vendors often supply a system that can include or exclude a significant fraction of the advertised "features." This reconfigurability can make performance modeling more difficult because of uncertainty about how the reconfiguration is accomplished and what the effect of each feature is upon performance. The other features generally supplied are multi-tasking, scheduling, exception handling, device drivers, mathematical and other libraries, compilation, linking and loading services, communication services, debugging tools, performance measurement tools, and a software development environment. These features are so numerous and varied that their effects on real-time performance and software reliability can only be reviewed on a case-by-case basis.

Two operating systems that have been reported in the literature are significant because they support a requirement that the resulting system shall be predictable. These two are Mars (Kopetz et al 1989) and the Spring Kernel (Stankovic and Ramamritham 1991). Mars is a statically scheduled system that has support for mechanized timing analysis and schedule generation. Mars is a distributed system that is linked together by Time Domain Multiplexed (TDM) data links (Preckshot 1993). The communication system is state-based (see Preckshot 1993) and state messages are unconsumed when read. Timestamps associate a "validity time" with each state message, and fault tolerance is assisted by not acting upon stale state data. A global time base is maintained within a small error window that permits absolute timing of events, causal ordering, accurate calculation of time intervals, and consistency between a real-time

database and the environment. The Mars system is designed to eliminate uncertainty at the expense of some flexibility.

The Spring Kernel is designed to allow somewhat more flexibility than the Mars system, but at the expense of predictability of some parts of the system. Real-time tasks are designated as "critical," "essential," and "non-essential." Critical tasks have bounded execution times and are statically scheduled. Essential and non-essential tasks are dynamically scheduled, with essential tasks forming a schedulable subset. The dynamic scheduler uses auxiliary information about tasks to enhance scheduling success. Imprecise calculations are used to limit essential task execution times where necessary. Non-essential tasks are delayed or aborted if insufficient time is available to complete all tasks to be scheduled.

The Mars system and the Spring Kernel illustrate two approaches to predictability using a real-time operating system. Commercially available real-time operating systems may have equivalently convincing methods of ensuring predictability, but demonstration of this depends upon the operating system vendor.

References

Auernheimer, Brent, Kemmerer, Richard A., September 1986, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 9, pp. 879–889.

Barter, R., and Zucconi, L., 1992, "Verification and Validation Techniques for Software Safety and Reliability," Lawrence Livermore National Laboratory, to be released.

Burns, Alan, and Wellings, Andy, 1990, *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Company.

Cabodi, G., Camurati, P., Prineto, P., Sonza Reorda, M., April 1991, "TPDL*: Extended Temporal Profile Description Language," *Software Practice and Experience*, Vol. 21(4), pp. 355–374.

Callison, H. Rebecca, Shaw, Alan C., April 1991, "Building a Real-Time Kernel: First Steps in Validating a Pure Process/Adt Model," *Software Practice and Experience*, Vol. 21(4), pp. 337–354.

Chang, Carl K., and four co-authors, March 1989, "Modeling a Real-Time Multitasking System in a Timed PQ Net," *IEEE Software*, pp. 46–51.

Clapp, Russel M., Duchesneau, Louis, Volz, Richard A., Mudge, Trevor N., Schultze, Timothy, August 1986, "Toward Real-Time Performance Benchmarks for ADA," *Comm. ACM 29*, No. 8, pp. 760–778.

Conte, Gianni, and Caselli, Stefano, 1991, Generalized Stochastic Petri Nets and Their Use in Modeling Distributed Architectures, *IEEE CH3001-5/91/0000/0296*, pp. 296–303.

Davis, Alan M., September 1992, "Operational Prototyping: A New Development Approach," *IEEE Software*, pp. 70–78.

Fidge, Colin, August 1991, "Logical Time in Distributed Computing Systems," *Computer*, pp. 28–33.

Gopinath, Prabha, Bihari, Thomas, Gupta, Rajiv, September 1992, "Compiler Support for Object-Oriented Real-Time Software," *IEEE Software*, pp. 45–50.

Haase, Volkmar H., September 1981, "Real-Time Behavior of Programs," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 5, pp. 494–501.

Hoare, C.A.R., October 1974, "Monitors: An Operating System Structuring Concept," *Comm. ACM 17*, 10, pp. 549–557.

Holt, R.C., 1983, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley Publishing Company.

Kenny, Kevin B., Lin, Kwei-Jay, May 1991, "Building Flexible Real-Time Systems Using the Flex Language," *Computer*, pp. 70–78.

Kopetz, H., and six co-authors, February 1989, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," I*EEE Micro*, pp. 25–40.

Kopetz, H., circa 1990, "Event-Triggered Versus Time-Triggered Real-Time Systems," Technische Universitat Wien, Institut Fur Technische Informatik.

Kopetz, H., Ochsenreiter, W., 1987, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. on Computers*, Vol. C-36, No. 8, pp. 933–940.

Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., Shutz, W., May 1991, "The Design of Real-Time Systems: From Specification to Implementation and Verification," *Software Engineering Journal*, pp. 72–82.

Lala, Jaynarayan H., Harper, Richard E., Alger, Linda S., May 1991, "A Design Approach for Ultrareliable Real-Time Systems," *Computer*, pp. 12–22.

Lamport, Leslie, July 1978, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM 21*, No. 7, pp. 558–565.

Lawrence, Dennis J., 1992, Subject: Software reliability guidance, Lawrence Livermore National Laboratory, to be released.

Lazowska, E.D., Zahorjan, J., Graham, G.S., and Sevcik, K.C., 1984, *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*, Prentice-Hall.

Leveson, Nancy G., Cha, Steven S., Shimeall, Timothy J., July 1991, "Safety Verification of Ada Programs Using Software Fault Trees," *IEEE Software*, pp 48–59.

Liu, C.L., Layland, James W., January 1973, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal ACM*, V. 20, No. 1, pp. 46–61.

MacDougall, M.H., 1987, *Simulating Computer Systems: Techniques and Tools*, MIT Press.

Natarajan, Swaminathan, Zhao, Wei, September 1992, "Issues in Building Dynamic Real-Time Systems," *IEEE Software*, pp. 16–21.

Oldehoeft, R.R., January 1983, "Program Graphs and Execution Behavior," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 1, pp. 103–108.

Park, Chang Yun, Shaw, Alan C., May 1991, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Computer*, pp. 48–57.

Peterson, James L., September 1977, "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3, pp. 223–252.

Pospischil, Gustav, Puschner, Peter, Vrchoticky, Alexander, Zainlinger, Ralph, September 1992, "Developing Real-Time Tasks with Predictable Timing," *IEEE Software*, pp. 35–44.

Preckshot, George G., 1993, "Data Communications Guidance," Lawrence Livermore National Laboratory, to be released.

Rosenblum, David S., May 1991, "Specifying Concurrent Systems with TSL," *IEEE Software*, pp. 52–61.

Sha, Lui, and Goodenough, John B., April 1990, "Real-Time Scheduling Theory and Ada," *Computer*, pp. 53–62.

Sha, Lui, Rajkumar, Ragunathan, and Lehoczky, John P., September 1990, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, Vol. 39, 9, pp. 1175–1185.

Shaw, Alan C., July 1989, "Reasoning About Time in Higher-Level Language Software," *IEEE Trans. on Software Engineering*, Vol. 15, No. 7, pp. 875–889.

Smith, Connie U., 1990, *Performance Engineering of Software Systems*, Addison-Wesley Publishing Co.

Stankovic, John A., October 1988, "Misconceptions About Real-Time Computing," *Computer*, pp. 10–19.

Stankovic, John A., Ramamritham, Krithi, May 1991, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, pp. 62–72.

Taylor, Tamara, Standish, Thomas A., December 1982, "Initial Thoughts on Rapid Prototyping Techniques," *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, pp. 160–166.

Warren, Carl, June 1991, "Rate Monotonic Scheduling," *IEEE Micro*, pp. 34 –38, 102.

Woodbury, Michael H., 1986, "Analysis of the Execution Time of Real-Time Tasks," *1986 Proc. IEEE Real-Time Systems Symposium*, pp. 89–96.

Young, Michal, Taylor, Richard N., October 1988, "Combining Static Concurrency Analysis with Symbolic Execution," *IEEE Trans. on Software Engineering*, Vol. 14, No. 10, pp. 1499–1511.

Zhao, Wei, Ramamritham, Krithivasan, and Stankovic, John A., May 1987a, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, 5, pp. 564 –577.

Zhao, Wei, Ramamritham, Krithivasan Stankovic, John A., August 1987b, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Trans. on Computers*, Vol. C-36, No. 8, pp. 949–960.

Zucconi, L., and Thomas, B., 1992, "Testing Existing Software for Safety-Related Applications," Lawrence Livermore National Laboratory, to be released.